



Monolithic Compiler Experiments Using C++ Expression Templates*

Lenore R. Mullin**
Edward Rutledge
Robert Bond

HPEC 2002
25 September, 2002
Lexington, MA

* This work is sponsored by the Department of Defense, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the Department of Defense.

MIT Lincoln Laboratory

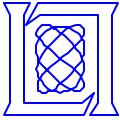
** Dr. Mullin participated in this work while on sabbatical leave from the Dept. of Computer Science, University of Albany, State University of New York, Albany, NY.

Report Documentation Page				Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE 25 SEP 2002		2. REPORT TYPE N/A		3. DATES COVERED -	
4. TITLE AND SUBTITLE Monolithic Compiler Experiments Using C++ Expression Templates*				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) MIT Lincoln Laboratory				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, distribution unlimited					
13. SUPPLEMENTARY NOTES The original document contains color images.					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 29	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

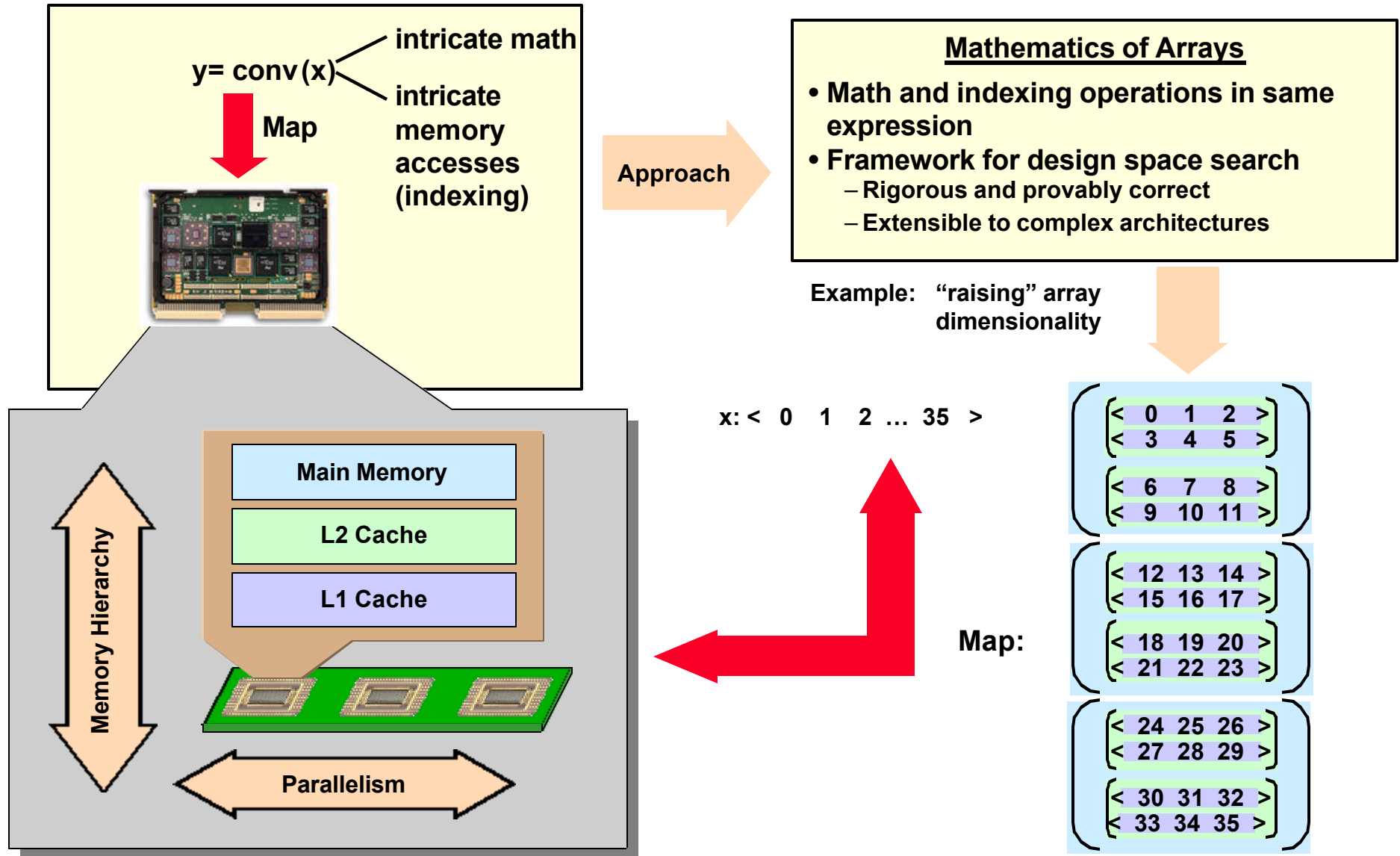


Outline

- ➔ • **Overview**
 - Motivation
 - The Psi Calculus
 - Expression Templates
- **Implementing the Psi Calculus with Expression Templates**
- **Experiments**
- **Future Work and Conclusions**

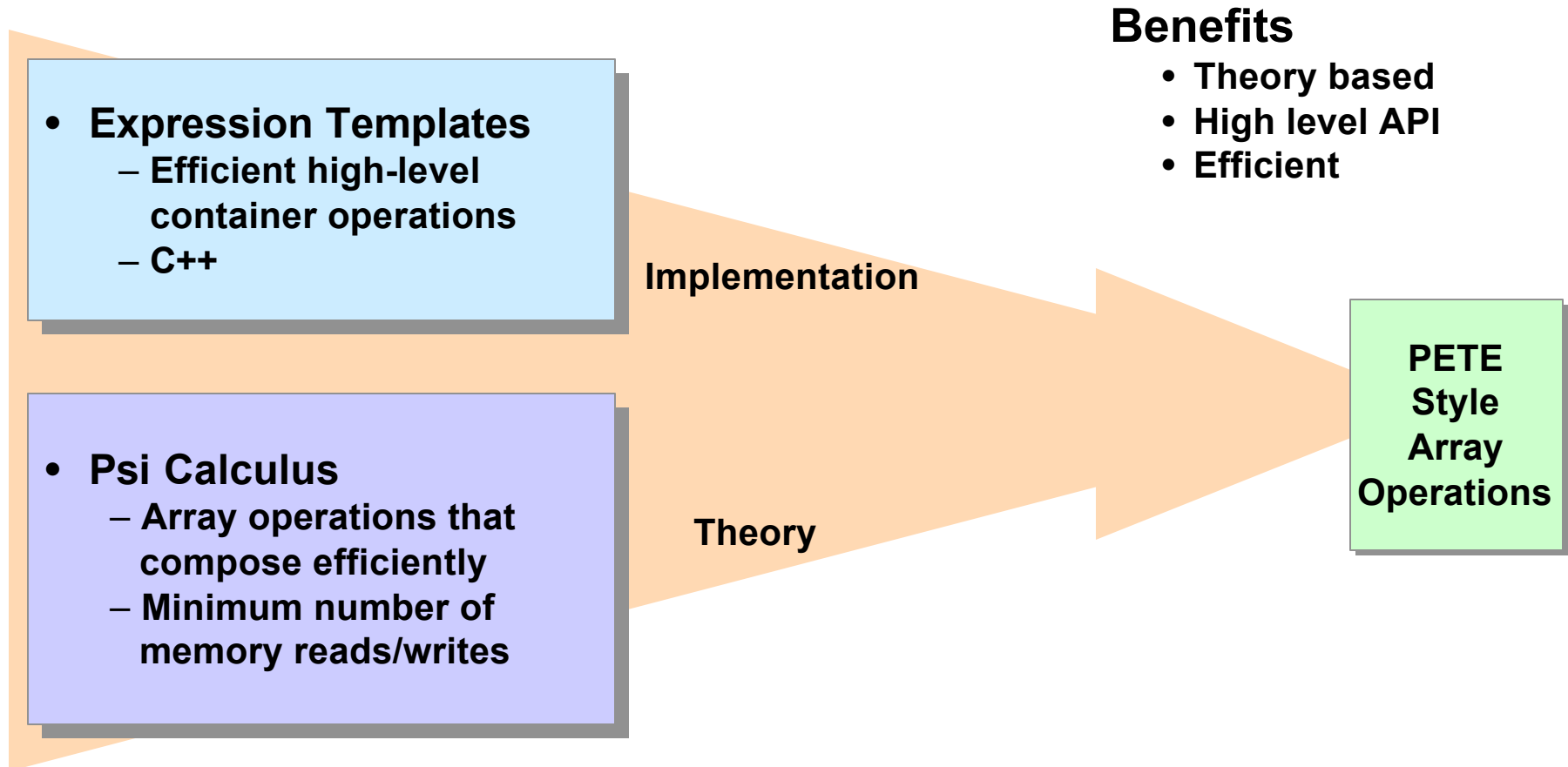


Motivation: The Mapping Problem

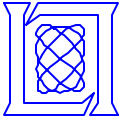




Basic Idea



Combining Expression Templates and Psi Calculus yields an optimal implementation of array operations



Psi Calculus¹ Key Concepts

Denotational Normal Form (DNF):

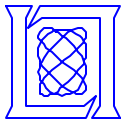
- Minimum number of memory reads/writes for a given array expression
- Independent of data storage order

Operational Normal Form (ONF):

- Like DNF, but takes data storage into account
- For 1-d expressions, consists of one or more loops of the form:
 $x[i]=y[\text{stride}*i+\text{offset}], l \leq i < u$
- Easily translated into an efficient implementation


Gamma function:
Specifies data
storage order

- Psi Calculus rules are applied mechanically to produce the DNF, which is optimal in terms of memory accesses
- The Gamma function is applied to the DNF to produce the ONF, which is easily translated to an efficient implementation



Some Psi Calculus Operations

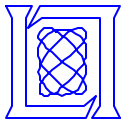
Operations	Arguments	Definition
take	<i>Vector A, int N</i>	Forms a Vector of the first N elements of A
drop	<i>Vector A, int N</i>	Forms a Vector of the last (A.size-N) elements of A
rotate	<i>Vector A, int N</i>	Forms a Vector of the last N elements of A concatenated to the other elements of A
cat	<i>Vector A, Vector B</i>	Forms a Vector that is the concatenation of A and B
unaryOmega	<i>Operation Op, dimension D, Array A</i>	Applies unary operator Op to D-dimensional components of A (like a for all loop)
binaryOmega	<i>Operation Op, Dimension Adim, Array A, Dimension Bdim, Array B</i>	Applies binary operator Op to Adim-dimensional components of A and Bdim-dimensional components of B (like a for all loop)
reshape	<i>Vector A, Vector B</i>	Reshapes B into an array having A.size dimensions, where the length in each dimension is given by the corresponding element of A
iota	<i>int N</i>	Forms a vector of size N, containing values 0 . . N-1

 = index permutation

 = operators

 = restructuring

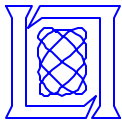
 = index generation



Convolution: Psi Calculus Decomposition

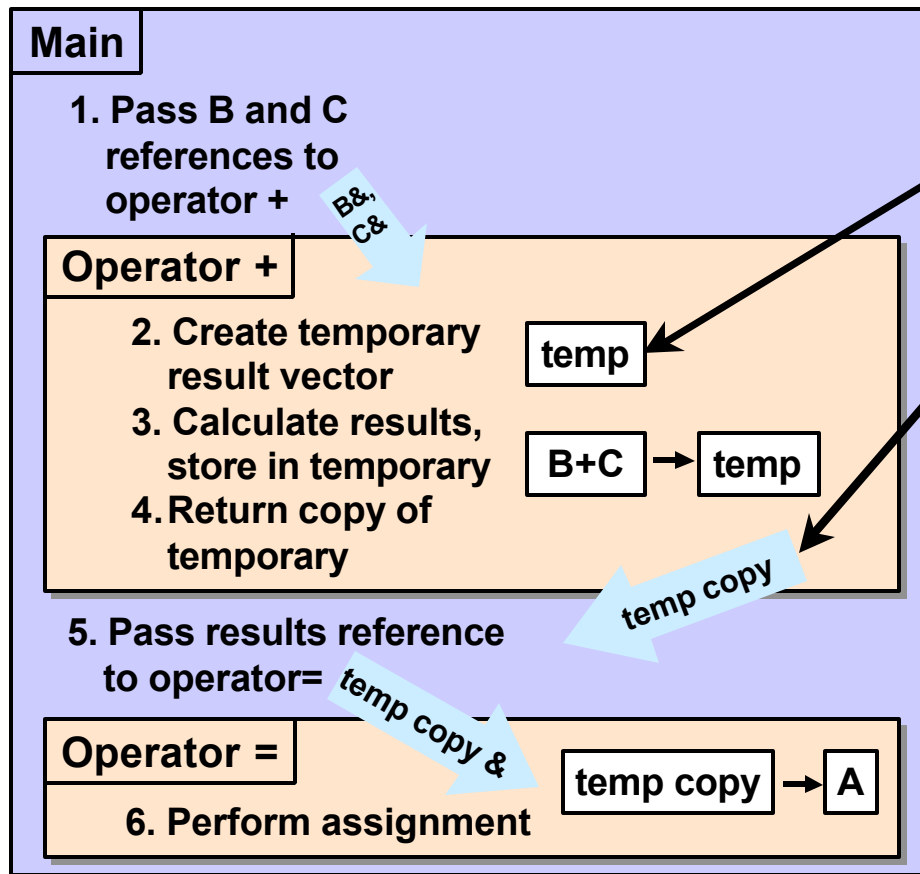
Definition of $y = \text{conv}(h, x)$	$y[n] = \sum_{k=0}^{M-1} h[k]x'[n-k]$ where x has N elements, h has M elements, $0 \leq n < N+M-1$, and x' is x padded by $M-1$ zeros on either end		
Algorithm and Psi Calculus Decomposition	Algorithm step	Psi Calculus	
	Initial step	$x = \langle 1 \ 2 \ 3 \ 4 \rangle$ $h = \langle 5 \ 6 \ 7 \rangle$	$x = \langle 1 \ 2 \ 3 \ 4 \rangle$ $h = \langle 5 \ 6 \ 7 \rangle$
	Form x'	$x' = \text{cat}(\text{reshape}(\langle k-1 \rangle, \langle 0 \rangle), \text{cat}(x, \text{reshape}(\langle k-1 \rangle, \langle 0 \rangle))) =$	$x' = \langle 0 \ 0 \ 1 \ \dots \ 4 \ 0 \ 0 \rangle$
	rotate x' ($N+M-1$) times	$x'_{\text{rot}} = \text{binaryOmega}(\text{rotate}, 0, \text{iota}(N+M-1), 1 \ x')$	$x'_{\text{rot}} = \begin{matrix} \langle 0 \ 0 \ 1 \ 2 \ \dots \rangle \\ \langle 0 \ 1 \ 2 \ 3 \ \dots \rangle \\ \langle 1 \ 2 \ 3 \ 4 \ \dots \rangle \\ \vdots \end{matrix}$
	take the “interesting” part of x'_{rot}	$x'_{\text{final}} = \text{binaryOmega}(\text{take}, 0, \text{reshape}(\langle N+M-1 \rangle, \langle M \rangle), 1, x'_{\text{rot}})$	$x'_{\text{final}} = \begin{matrix} \langle 0 \ 0 \ 1 \rangle \\ \langle 0 \ 1 \ 2 \rangle \\ \langle 1 \ 2 \ 3 \rangle \\ \vdots \end{matrix}$
	multiply	$\text{Prod} = \text{binaryOmega}(*, 1, h, 1, x'_{\text{final}})$	$\text{Prod} = \begin{matrix} \langle 0 \ 0 \ 7 \rangle \\ \langle 0 \ 6 \ 14 \rangle \\ \langle 5 \ 12 \ 21 \rangle \\ \vdots \end{matrix}$
	sum	$Y = \text{unaryOmega}(\text{sum}, 1, \text{Prod})$	$Y = \langle 7 \ 20 \ 38 \ \dots \rangle$

Psi Calculus reduces this to DNF with minimum memory accesses



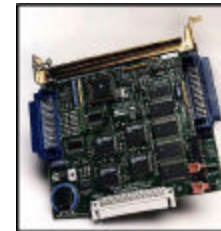
Typical C++ Operator Overloading

Example: $A=B+C$ vector add



2 temporary vectors created

Additional Memory Use

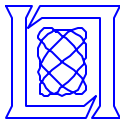


- Static memory
- Dynamic memory (also affects execution time)

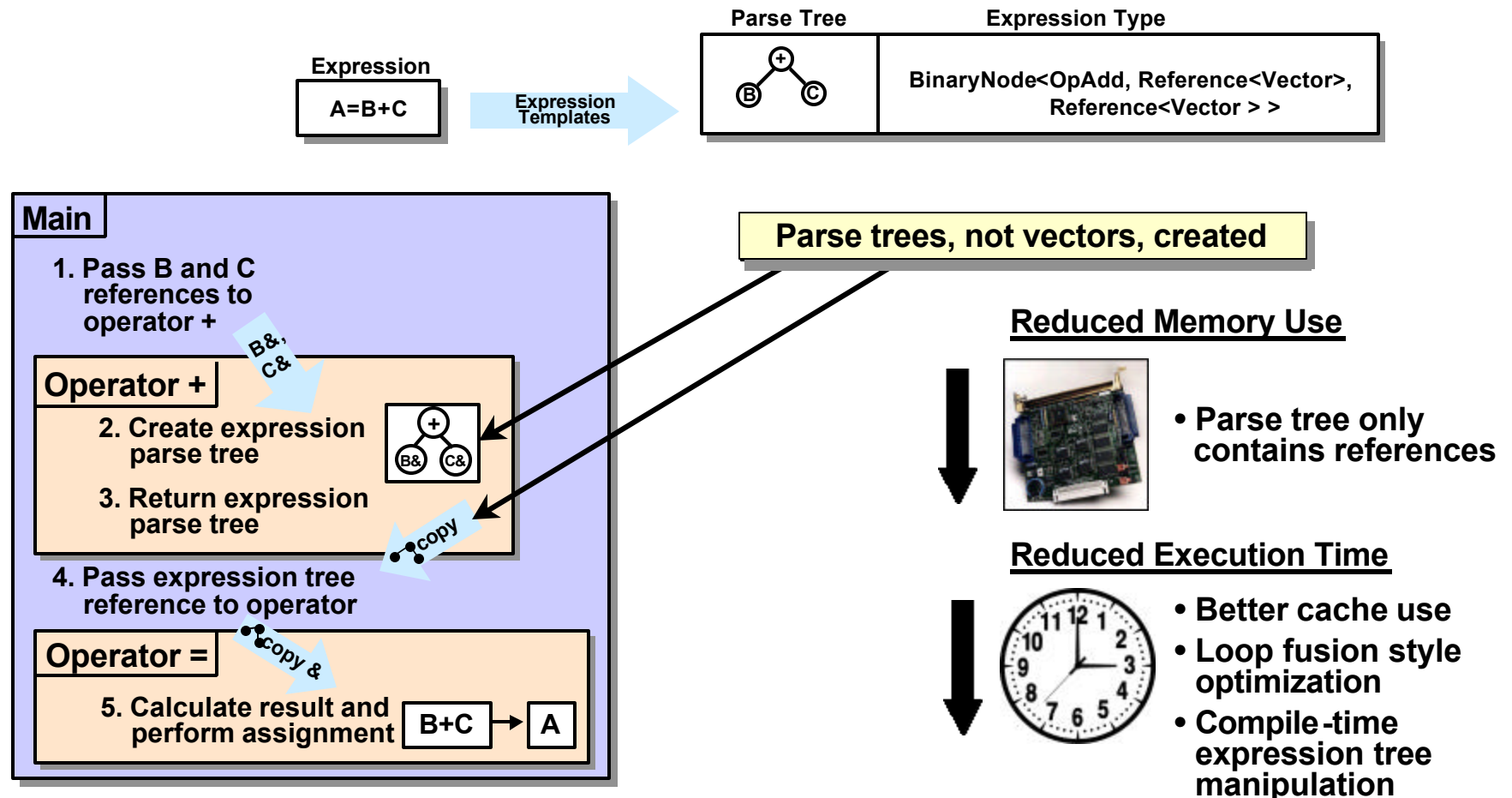
Additional Execution Time



- Cache misses/page faults
- Time to create a new vector
- Time to create a copy of a vector
- Time to destruct both temporaries

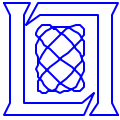


C++ Expression Templates and PETE



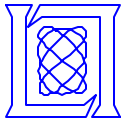
- PETE, the Portable Expression Template Engine, is available from the Advanced Computing Laboratory at Los Alamos National Laboratory
- PETE provides:
 - Expression template capability
 - Facilities to help navigate and evaluating parse trees

PETE: <http://www.acl.lanl.gov/pete>



Outline

- Overview
 - Motivation
 - The Psi Calculus
 - Expression Templates
- ➔ • **Implementing the Psi Calculus with Expression Templates**
- Experiments
- Future Work and Conclusions



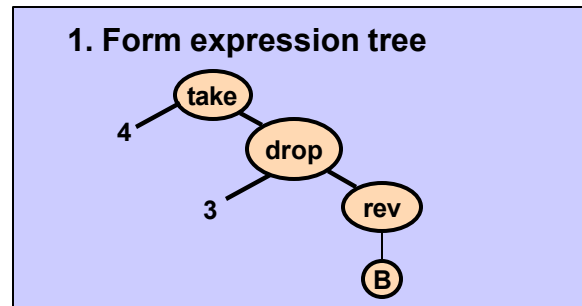
Implementing Psi Calculus with Expression Templates

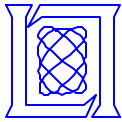
Example:

A=take(4,drop(3,rev(B)))

B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>





Implementing Psi Calculus with Expression Templates

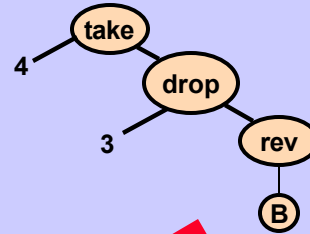
Example:

A=take(4,drop(3,rev(B)))

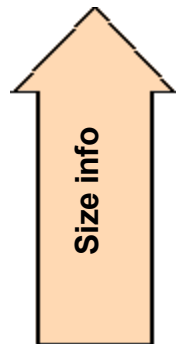
B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

1. Form expression tree



2. Add size information



size=10 B



Implementing Psi Calculus with Expression Templates

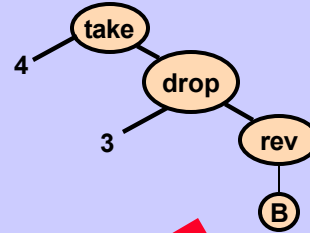
Example:

A=take(4,drop(3,rev(B)))

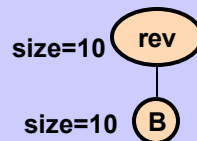
B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

1. Form expression tree



2. Add size information



Size info



Implementing Psi Calculus with Expression Templates

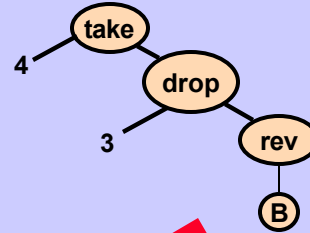
Example:

A=take(4,drop(3,rev(B)))

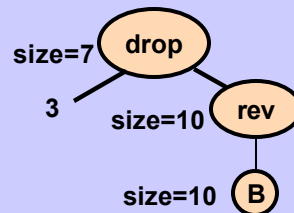
B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

1. Form expression tree



2. Add size information



Size info



Implementing Psi Calculus with Expression Templates

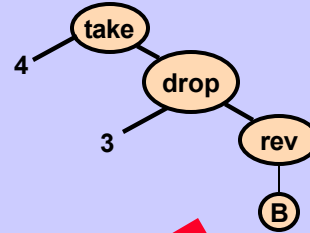
Example:

A=take(4,drop(3,rev(B)))

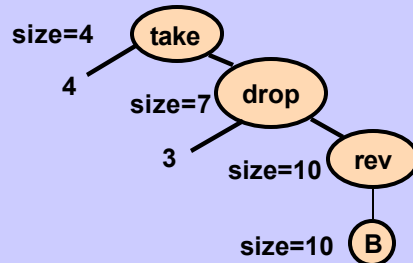
B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

1. Form expression tree



2. Add size information



Size info



Implementing Psi Calculus with Expression Templates

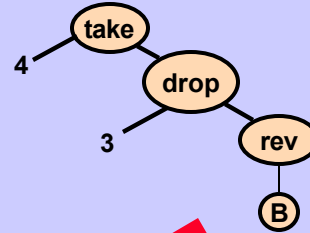
Example:

$A = \text{take}(4, \text{drop}(3, \text{rev}(B)))$

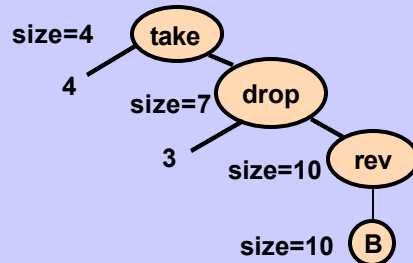
$B = \langle 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \rangle$

$A = \langle 7\ 6\ 5\ 4 \rangle$

1. Form expression tree



2. Add size information

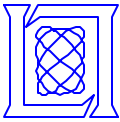


3. Apply Psi Reduction rules

size=10

$A[i] = B[i]$

Reduction



Implementing Psi Calculus with Expression Templates

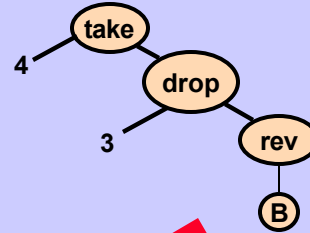
Example:

A=take(4,drop(3,rev(B)))

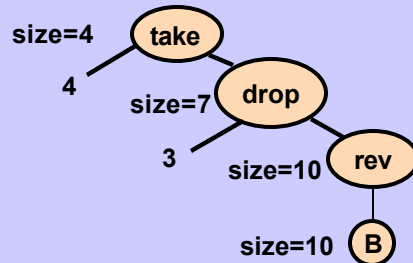
B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

1. Form expression tree



2. Add size information



3. Apply Psi Reduction rules

size=10 $A[i] = B[-i+B.size-1]$
 $= B[-i+9]$

size=10 $A[i] = B[i]$

Reduction



Implementing Psi Calculus with Expression Templates

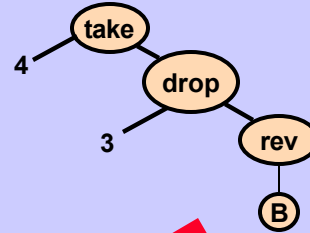
Example:

$A = \text{take}(4, \text{drop}(3, \text{rev}(B)))$

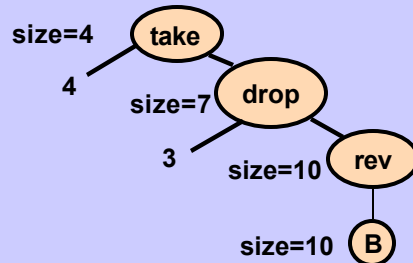
$B = \langle 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \rangle$

$A = \langle 7\ 6\ 5\ 4 \rangle$

1. Form expression tree



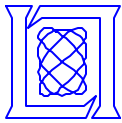
2. Add size information



3. Apply Psi Reduction rules

size=7	$A[i] = B[-(i+3)+9]$ $= B[-i+6]$
size=10	$A[i] = B[-i+B.size-1]$ $= B[-i+9]$
size=10	$A[i] = B[i]$

Reduction



Implementing Psi Calculus with Expression Templates

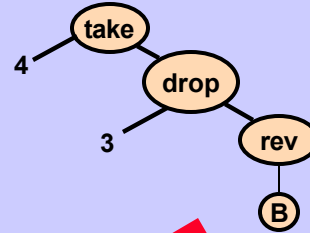
Example:

A=take(4,drop(3,rev(B)))

B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

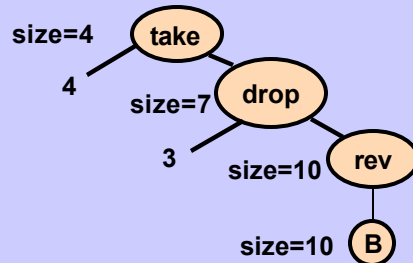
1. Form expression tree



Recall:

**Psi Reduction for 1-d arrays
always yields one or more
expressions of the form:**
 $x[i]=y[\text{stride} \cdot i + \text{offset}]$
 $l \leq i < u$

2. Add size information



3. Apply Psi Reduction rules

size=4	$A[i]=B[-i+6]$
size=7	$A[i]=B[-(i+3)+9]$ $=B[-i+6]$
size=10	$A[i]=B[-i+B.\text{size}-1]$ $=B[-i+9]$
size=10	$A[i]=B[i]$

Size info

Reduction



Implementing Psi Calculus with Expression Templates

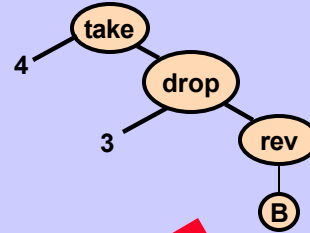
Example:

A=take(4,drop(3,rev(B)))

B=<1 2 3 4 5 6 7 8 9 10>

A=<7 6 5 4>

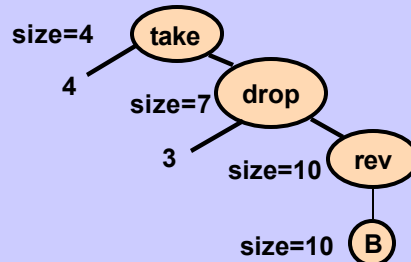
1. Form expression tree



Recall:

Psi Reduction for 1-d arrays
always yields one or more
expressions of the form:
 $x[i]=y[\text{stride} \cdot i + \text{offset}]$
 $l \leq i < u$

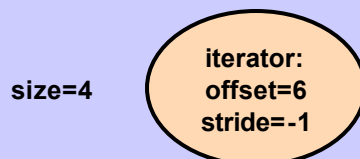
2. Add size information



3. Apply Psi Reduction rules

size=4	$A[i]=B[-i+6]$
size=7	$A[i]=B[-(i+3)+9]$ $=B[-i+6]$
size=10	$A[i]=B[-i+B.\text{size}-1]$ $=B[-i+9]$
size=10	$A[i]=B[i]$

4. Rewrite as sub-expressions with iterators at the leaves, and loop bounds information at the root

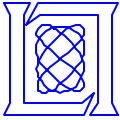


- Iterators used for efficiency, rather than recalculating indices for each i
- One “for” loop to evaluate each sub-expression



Outline

- Overview
 - Motivation
 - The PSI Calculus
 - Expression Templates
- Implementing the Psi Calculus with Expression Templates
- ➔ • **Experiments**
- Future Work and Conclusions

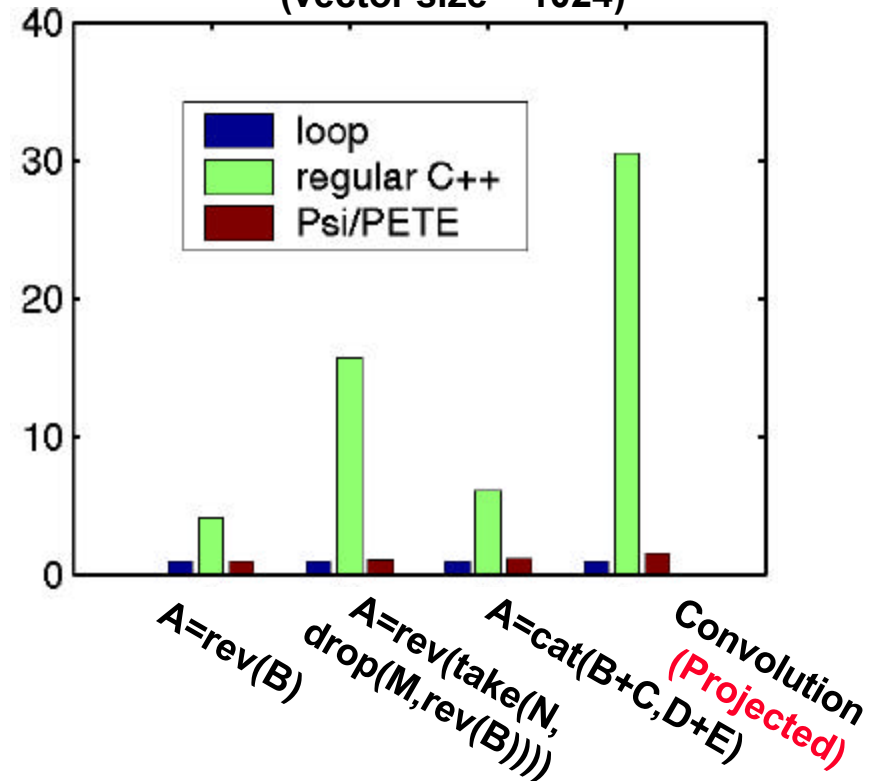


Experiments

Results

- Loop implementation achieves good performance, but is problem specific and low level
- Traditional C++ operator implementation is general and high level, but performs poorly when composing many operations
- PETE/Psi array operators perform almost as well as the loop implementation, compose well, are general, and are high level

Execution Time Normalized to Loop Implementation
(vector size = 1024)



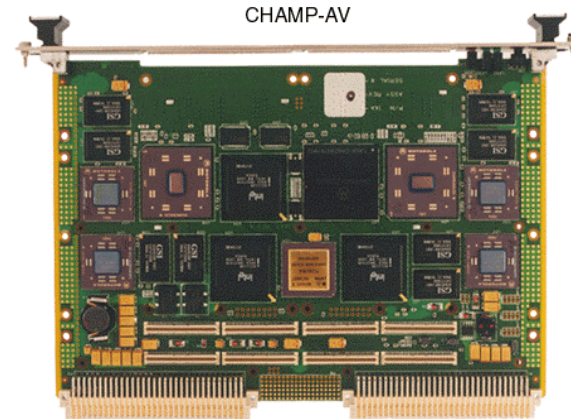
Test ability to compose operations



Experimental Platform and Method

Hardware

- **DY4 CHAMP-AV Board**
 - Contains 4 MPC7400's and 1 MPC 8420
- **MPC7400 (G4)**
 - 450 MHz
 - 32 KB L1 data cache
 - 2 MB L2 cache
 - 64 MB memory/processor



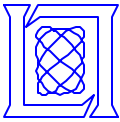
Software

- **VxWorks 5.2**
 - Real-time OS
- **GCC 2.95.4 (non-official release)**
 - GCC 2.95.3 with patches for VxWorks
 - Optimization flags:
-O3 -funroll-loops -fstrict-aliasing

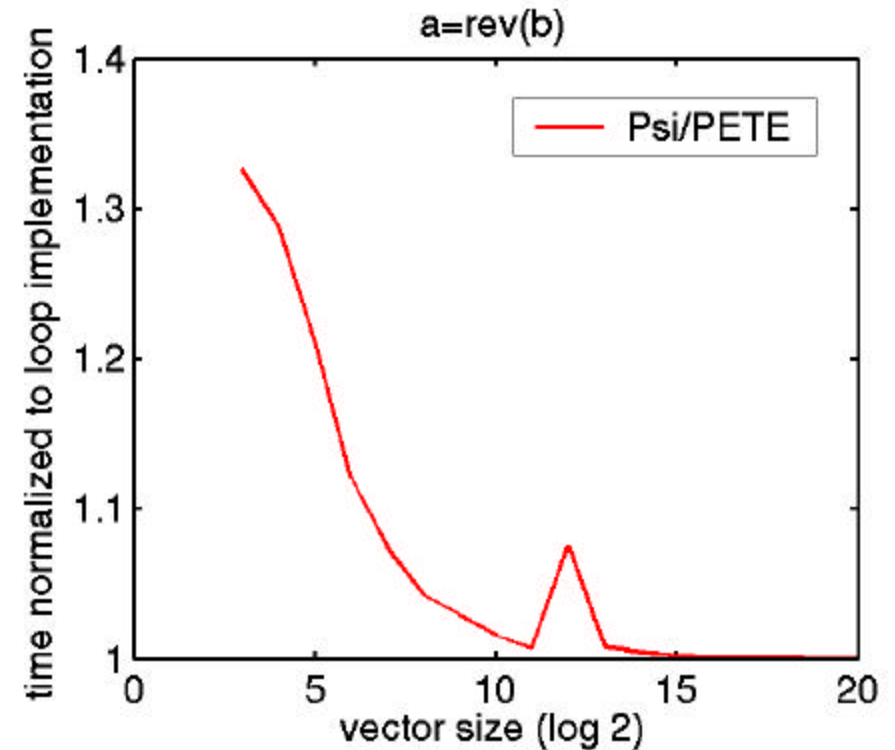
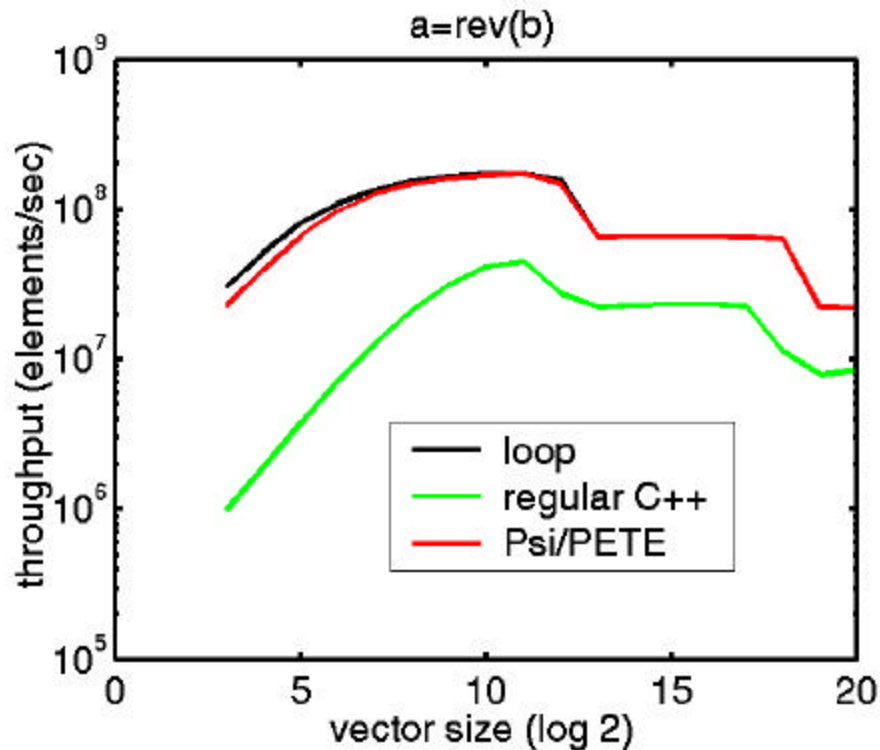
Method

- Run many iterations, report average, minimum, maximum time
 - From 10,000,000 iterations for small data sizes, to 1000 for large data sizes
- All approaches run on same data
- Only average times shown here
- Only one G4 processor used

- Use of the VxWorks OS resulted in very low variability in timing
- High degree of confidence in results



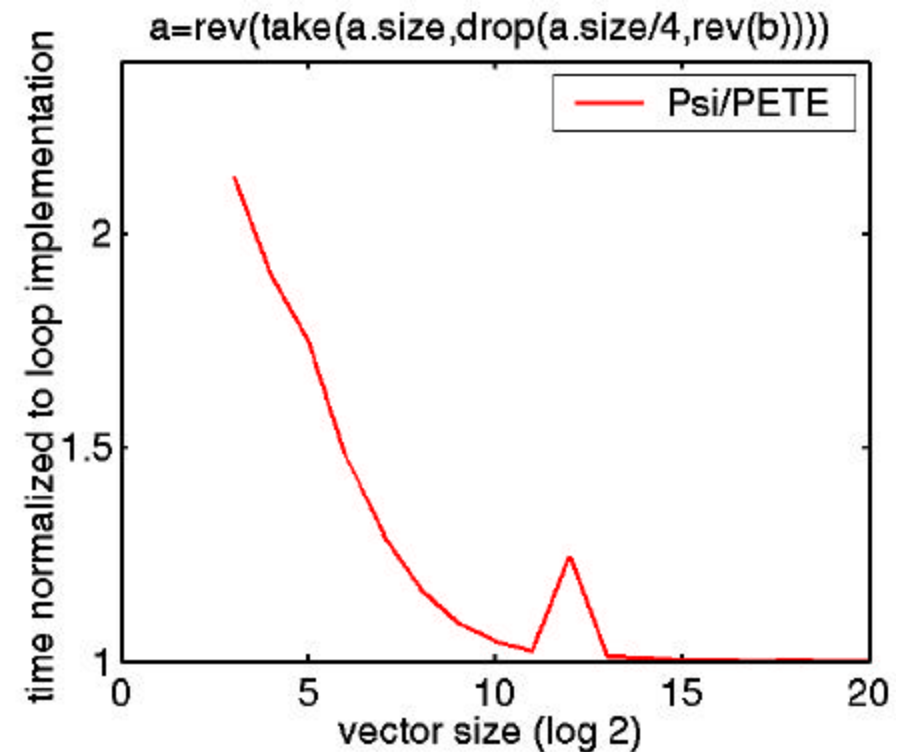
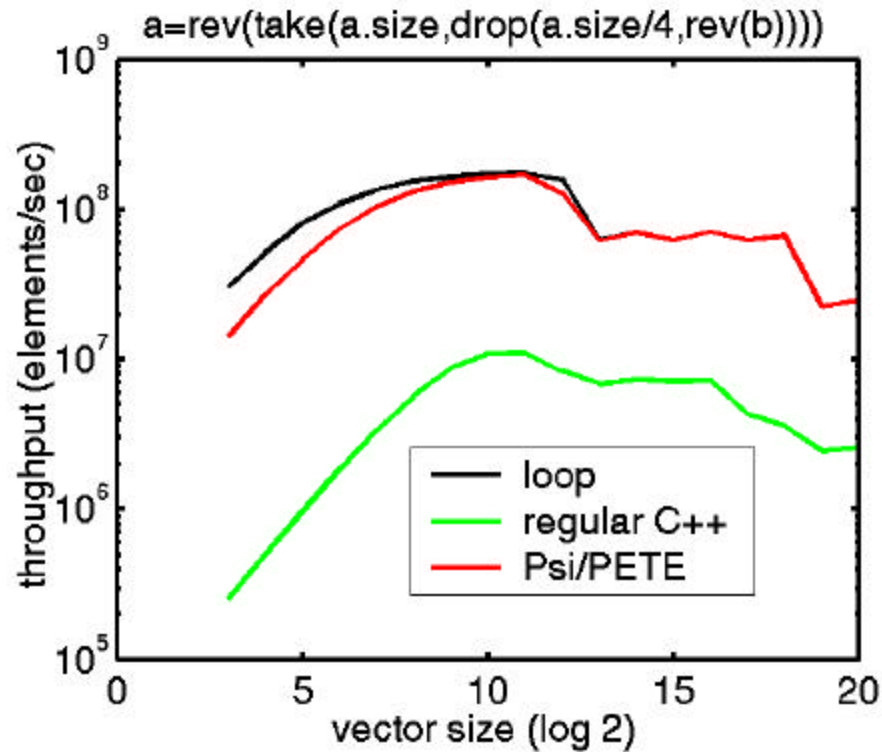
Experiment 1: $A = \text{rev}(B)$



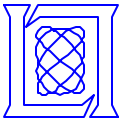
- PETE/Psi implementation performs nearly as well as hand coded loop, and much better than regular C++ implementation
- Some overhead associated with expression tree manipulation



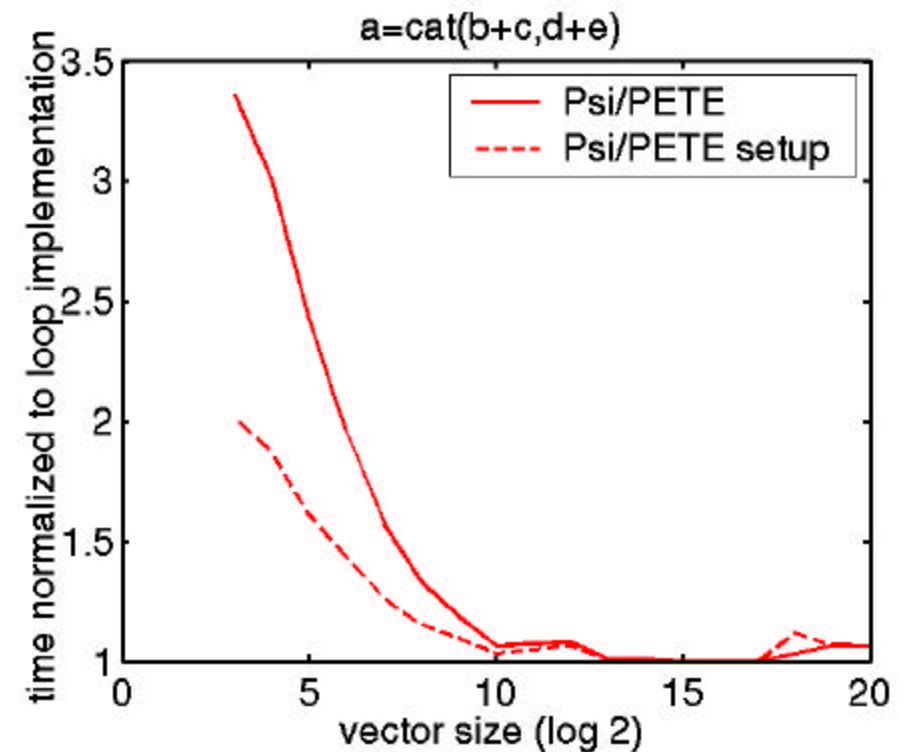
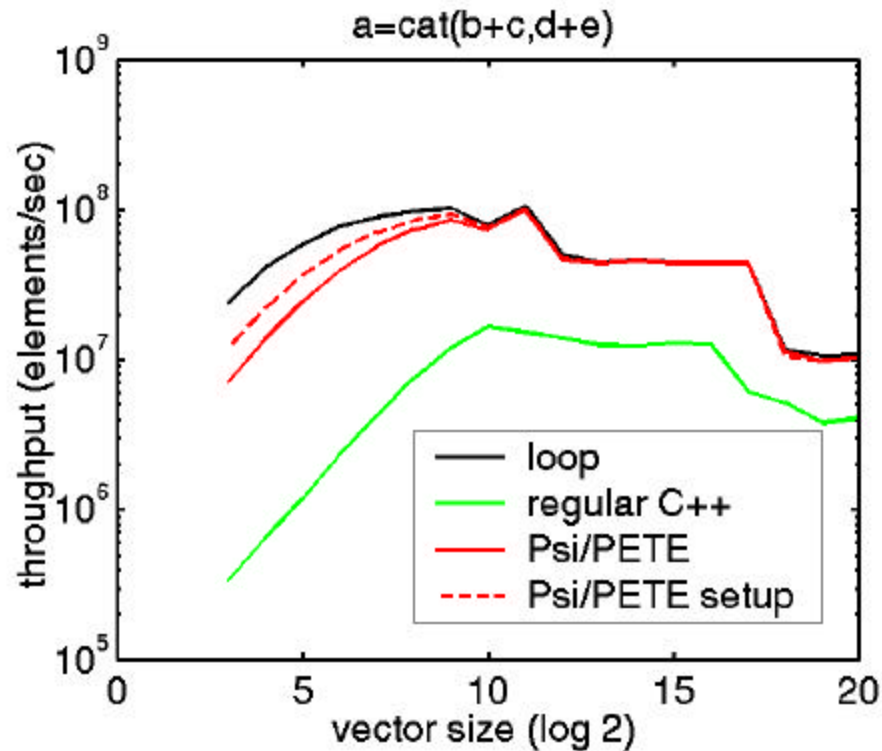
Experiment 2: $a = \text{rev}(\text{take}(N, \text{drop}(M, \text{rev}(b))))$



- Larger gap between regular C++ performance and performance of other implementations → regular C++ operators do not compose efficiently
- Larger overhead associated with expression-tree manipulation due to more complex expression



Experiment 3: $a = \text{cat}(b+c, d+e)$



- Still larger overhead associated with tree manipulation due to `cat()`
- Overhead can be mitigated by “setup” step prior to assignment



Outline

- **Overview**
 - Motivation
 - The PSI Calculus
 - Expression Templates
- **Implementing the PSI Calculus with Expression Templates**
- **Experiments**
- ➔ • **Future Work and Conclusions**



Future Work

- **Multiple Dimensions:** Extend this work to N-dimensional arrays (N is any non-negative integer)
- **Parallelism:** Explore dimension lifting to exploit multiple processors
- **Memory Hierarchy:** Explore dimension lifting to exploit levels of memory
- **Mechanize Index Decomposition:** Currently a time consuming process done by hand
- **Program Block Optimizations:** PETE-style optimizations across statements to eliminate unnecessary temporaries



Conclusions

- **Psi calculus provides rules to reduce array expressions to the minimum of number of reads and writes**
- **Expression templates provide the ability to perform compiler preprocessor-style optimizations (expression tree manipulation)**
- **Combining Psi calculus with expression templates results in array operators that**
 - **Compose efficiently**
 - **Are high performance**
 - **Are high level**
- **The C++ template mechanism can be applied to a wide variety of problems (e.g. tree traversal ala PETE, graph traversal, list traversal) to gain run-time speedup at the expense of compile time/space**